

Object-oriented Design Patterns and System Dynamics Components

By

Dr. Warren W. Tignor, Ph.D.

*Vice President, Kimmich Software Systems, Inc.
7235 Dockside Lane
Columbia, Maryland 21045 USA*

Magne Myrtveit

*Founder and Senior Vice President
Powersim AS, Hellandsneset, N-5936 Manger, Norway
Telephone: +47 88 02 34 34 – Facsimile: +47 56 37 35 00
E-mail: magne.myrtveit@powersim.no
Web page: <http://www.powersim.no>*

1 Abstract

The software engineering community uses an Object-Oriented Analysis and Design (OOAD) methodology to define, design and build software systems. The tools and trade of System Dynamics is heavily dependent upon software to successfully model and solve problems. This paper explores the Object-Oriented concepts of “patterns” and “classes” and how they relate to System Dynamic “models”, “components”, “molecules”, and “archetypes”. Specific examples will be discussed with similarities and differences as well as strengths and weaknesses and areas of application.

In the Object-Oriented world, design patterns capture generic solutions that have developed and evolved over time and describe them as structures or objects for reuse. These solutions are the subject of untold redesign and re-coding as software engineers have struggled for greater reuse and flexibility in code. Some design patterns can be used “as is” to form solutions or parts of solutions, while others serve as generic templates that can be refined into concrete solutions.

The term component (cf. Myrtveit 2000) is used for a model “class” that can serve as a building block when creating model “objects”. Components have interfaces defining the variables that carry information between the components and the rest of the model. Design patterns can be used both to implement and to document components.

2 Introduction

Expert designers, regardless of field of expertise, know not to solve every problem from first principles, Vlissides, J., Helm, R., Johnson, R., & Gamma, E. (1995). It is beneficial to reuse solutions that have worked in the past. When designers find a good solution, they us it over and

over; this is what makes them experts, Vlissides et al., (1995). A designer familiar with patterns can apply them to new problems without having to discover them.

Patterns have been long recognized in other disciplines as important in crafting complex systems, Vlissides et al., (1995). Christopher Alexander and his colleagues were probably the first to propose the idea of using a pattern language to architect buildings and cities, Vlissides et al., (1995). Alexander, C. et al (1977) said, “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”, (page *x*). Alexander et al. (1977) were talking about buildings and towns; but what he says is true of software object-oriented design, Vlissides et al., (1995).

Regarding patterns, this paper examines the relationships of software object-oriented design and System Dynamics software model design and architectures. Forrester (1990) set the “cornerstone” structures of System Dynamics. Bruner (1960) suggests that understanding the structure of a subject is essential to understanding the subject. This paper will show that an understanding of the fundamental System Dynamics structures will allow other things, Design Patterns in the Object-Oriented sense, to be related meaningfully. For example, the tools and trade of System Dynamics are heavily dependent upon software to successfully model and solve problems. Since the software engineering community uses an Object-oriented Analysis and Design (OOAD) methodology to define, design and build software systems, the same methodology may apply to System Dynamics.

Patterns are fundamental to object-oriented design. A pattern to one person may be a primitive building block to another. For this paper, the point of view of a pattern is as descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context, Vlissides et al., (1995). To this end, the paper explores the object-oriented concepts of “patterns” and “classes” and how they relate to System Dynamic “models”, “components”, “molecules”, and “archetypes”. Specific examples will be discussed with similarities and differences as well as strengths and weaknesses and areas of application.

3 Statement of the Problem

The paper proposes that object-oriented software design patterns are applicable to the field of System Dynamics and the construction of software simulation models. The hypothesis is that just as design patterns make it easier to reuse successful object-oriented designs and architecture, they can help the System Dynamist reuse simulation model designs and architectures.

4 Literature Review

The literature review is organized by the following categories: System Dynamics, Object-Oriented Design, Design Patterns and Applied Systems. The first three categories are intended to provide a basis of comparison using foundation statements about each of the disciplines (System Dynamics, Object-Oriented Design, and Design Patterns). The Applied Systems category represents a survey of literature using one or more of the foundation technologies (System Dynamics, Object-Oriented Design, and Design Patterns).

4.1 System Dynamics Background

Forrester set the cornerstone for the structure of System Dynamics and it has stood the test of time. In Principles of Systems, Forrester (1990) states that structure is essential if we are to effectively interrelate and interpret our observations in any field of knowledge: “Without an organizing structure, knowledge is a mere collection of observations, practices, and conflicting incidents” (p. 1-2). It is the structure of a subject that guides us in organizing information. “If one knows a structure or pattern on which he can depend, it helps him to interpret his observations. An observation may at first seem meaningless, but knowing that it must fit into one of a limited number of categories helps in the identification. Structure exists in many layers or hierarchies. Within any structure there can be substructures”, (Forrester, 1990, p. 4-1).

Likewise, Bruner (1960) tells us that it is the understanding of the structure of a subject that allows many other things to be related meaningfully. Bruner (1960) tells us that learning through the transfer of principles is dependent upon mastery of the structure of a subject. Understanding the fundamentals makes a subject comprehensible. Human memory is dependent upon structured patterns for recall. Understanding the specific case of a structure is a model for understanding other things like it that one may encounter. The constant reexamination of material’s structure results in a narrowing of the gap between advanced and elementary knowledge.

Forrester (1990) established the cornerstones of System Dynamic structure as the following: the closed boundary; feedback loops; levels and rates; and, within a rate, the goal, apparent condition, discrepancy, and action. Figure 1 presents these structure elements in hierarchical form (Forrester, 1990, p. 4-1):

- The closed system generating behavior within a boundary
 - The feedback loop
 - Levels as one fundamental variable type
 - Rates as the other fundamental variable type
 - The goal as one component of a rate
 - The apparent condition against which the goal is compared
 - The discrepancy between goal and apparent condition
 - The action resulting from the discrepancy.

Figure 1: System Dynamics Structure

The essential idea in a closed boundary system focuses on the interactions that produce growth, fluctuation, and change within the system. The boundary, Figure 2, “...encompasses the smallest number of components, within which the dynamic behavior under study is generated”, (Forrester, 1990, p. 4-2).

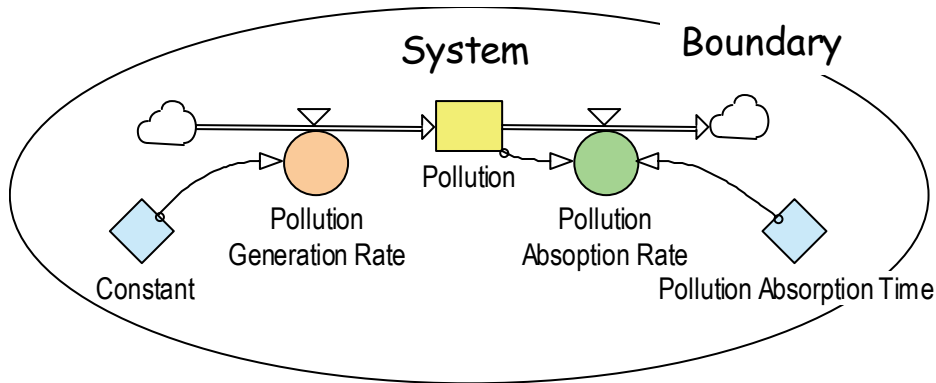


Figure 2: System Dynamics Boundary Concept

The feedback loop is the basic building block within the system boundary. The feedback loop couples the path connecting decision, action, system level, and information about the system level, with the final connection returning to the decision point, Figure 3.

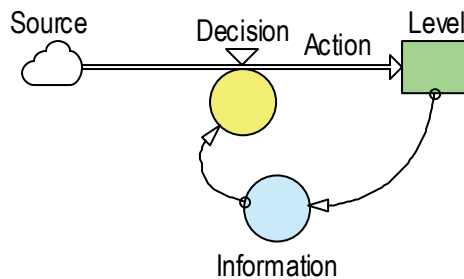


Figure 3: System Dynamics Feedback Loop

In this context, a decision process controls system action. The decision uses the available information to control action that influences the system level, and new information arises to modify the decision stream (Forrester, 1990).

Interconnecting feedback loops or a single feedback loop may constitute a system. At a lower level, feedback loops contain a substructure. Forrester (1990) tells us that there are two types of fundamental variable elements within each loop: levels and rates.

The level variable describes the condition of the system at any particular time. The level accumulates the results of action within the system. “The level variable accumulates the flows described by the rate variables”, (Forrester, 1990, p.4-5).

In contrast to levels, the rate variable tells how fast the levels are changing. “The rate equations are the policy statements that describe action in a system, that is, the rate equations state the action output of a decision point in terms of the information inputs to that decision”, (Forrester, 1990, p. 4-6).

Lastly, there is a substructure within a rate. According to Forrester (1990), there are four concepts within a rate equation: 1. Goal, 2. Observed condition of the system, 3. Discrepancy between goal and observed condition and 4. Statement of how action is to be based on the discrepancy, (p. 4-14), see Figure 4.

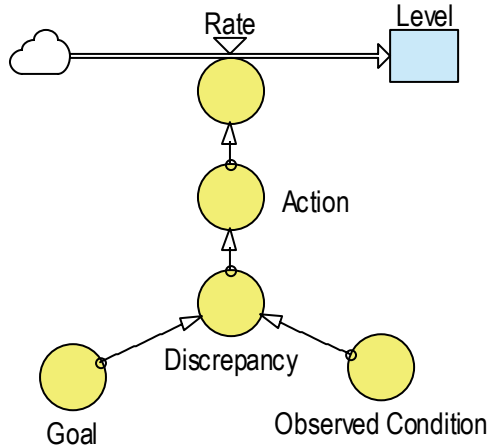


Figure 4: System Dynamics Rate Components

Forrester (1990) clarified that computing the successive time steps in the dynamic behavior of a system needed a standardized sequence for the computation and a terminology to use in designating the procedure. He illustrated the computation in time steps as shown in Figure 5 below:

	← DT →				
	R20.JK				
L2.J	R2I.JK				
L1.J	R1.JK				
5	5+DT	5+2DT	5+3DT	6	
J	K	L			
Time →					

Figure 5: System Dynamic Time Sequence

The time sequence figure assumes that the computations at time 5 were completed and the next computation period is 5+DT. The abbreviation DT stands for “difference in time”, Forrester (1990). “The 5 and the 6 in the figure represent the units of time used in defining the system, for example, weeks or months, but the appropriate solution interval need not be the same as the unit of time measurement”, (Forrester, 1990, p. 5-1).

The figure illustrated that there were four computations of system condition in each unit of time; “K” designates the current period of time and “J” the previous period of time. The levels L1.J and L2.J designate two values, system states, at the time “J”. The rate R1.JK flowed into level L1. The rate R2I.JK flowed into level L2 while the rate R20.JK flowed out of level L2, Forrester, (1990).

4.2 Object-Oriented Background

The objected-oriented paradigm captures system and software engineering work product in frameworks of packages, classes, objects, and methods. The language of the customer is captured by Use Cases as a statement of requirement and concept of operation. Leveraging the object-oriented paradigm to System Dynamics models may lead to benefits such as better understood: models, software design, and reusable software model libraries

Interestingly, our colleagues in the discrete simulation and modeling world have already recognized the opportunity to use object-oriented design and describe it with the Universal Modeling Language (UML). Braude (1998) applied recent advances in object-oriented research to propose a “class-level” framework for discrete simulations. Schöckle (1994) took an “object-oriented” approach in his work with modeling systems.

Braude (1998) says that there has been relatively little sharing of code or design for discrete simulation systems. Sharing, if any, has typically occurred at the tool level by means of commercially available graphics-based environments for building simulations (Braude, 1998).

Based on the maturity of the object-oriented paradigm and the adoption of UML, Braude (1998) believes that the time has arrived to attempt to design a standard framework for discrete simulations. He cites the definition of an application framework as a reusable, “semicomplete” application that can be specialized to produce custom applications, (Fayad & Schmidt, 1997).

Schöckle (1994) says that the object-oriented paradigm offers several possibilities not available in the traditional procedural programming approach, which help to deal with complex systems:

1. Object-oriented building blocks are “objects” which encapsulate functions and data.
2. Procedural building blocks are “procedures” which only abstract their functions.

Additionally, the object-oriented design provides concepts for managing complexity not available in procedural environments: classes, inheritance, polymorphism, and communication of messages (Schöckle 1994).

4.3 Fundamental Object-Oriented Structures

Taylor (1990) identifies three keys to understanding the object-oriented paradigm, i.e., objects, messages, and classes. According to Taylor (1990), the concept of software objects came from the need to model real-world objects in computer simulations. For example, SIMULA, created by O. J. Dahl and Kristen Nygaard of Norway, builds accurate working models of complex physical systems containing thousands of objects, Taylor (1990).

An object is software that contains a collection of related procedures and data, (Taylor, 1990). In the object-oriented approach, procedures go by a special name; they are called methods. In keeping with traditional programming terminology, the data elements are referred to as variables because their values can change over time, (Taylor, 1990).

Real-world objects can have an unlimited number of effects on each other, e.g., create, destroy, lift, attach, buy, sell. The way objects interact is by sending messages to each other. “A message is simply the name of an object followed by the name of a method the object knows how to execute, (Taylor, 1990, p. 19)”. Taylor (1990) adds that if a method requires any additional information in order to execute, the message includes that information as a collection of data elements called parameters.

Since most software systems or simulations will have a plethora of objects, methods, and variables as opposed to a single object with its methods and variables, the concept of class was created. “A class is a template that defines the methods and variables to be included in a particular type of object. The descriptions of the methods and variables that support them are included only once, in the definition of the class. The objects that belong to a class, called

instances of the class, contain only their particular values for the variables, (Taylor, 1990, p. 20)”.

When Grady Booch, Ivar Jacobson and James Rumbaugh began crafting the Unified Modeling Language, they aimed to produce a standard means of expressing design that would reflect the best practices of industry, and also demystify the process of software system modeling (Fowler & Scott, 1997). They believed that the availability of a standard modeling language would encourage developers to model their software systems before building them (Fowler & Scott, 1997).

Fowler says that one of the biggest challenges in software development is building the “right” system that meets the customer’s needs at a reasonable price. To Fowler and Scott (1997), achieving good communication with the customer, and an understanding of the customer’s world is key to developing good software. To this end, Fowler and Scott recommend the object-oriented Use Case, a snapshot of one aspect of a system requirement.

4.4 Design Patterns

Design patterns describe the key ideas in the system, Fowler and Scott (1997). Patterns help explain why a design is the way it is. The design pattern represents the fundamental algorithm being implemented by the software, an algorithm that is repeated in many other designs. Vlissides et al., (1995) characterize design patterns as a description of communicating objects and classes that are customized to solve a general design problem in a particular context. The design pattern names, abstracts and identifies the key aspects of a common design structure that makes it useful for creating a reusable object-oriented design. Design patterns describe simple and elegant solutions to specific problems, Vlissides et al., (1995). Design patterns capture designs that have developed and evolved over time; they reflect extensive redesign and recoding as developers have striven for greater reuse and flexibility in their software, Vlissides et al., (1995).

Designing object-oriented software is considered hard work; making it reusable is even harder. Vlissides et al., (1995) says that one has to find the pertinent objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationships among them. The design needs to be specific to the problem at hand but also general enough to address future problems and requirements. Redesign is to be avoided if possible and minimized at the least.

Vlissides et al., (1995) say that a pattern has four essential elements:

1. The Pattern Name describes a design problem, its solutions, and consequences in a word or two. The name allows one to design at a higher level of abstraction. The vocabulary of pattern names facilitates dialog. The name enables thinking about good designs and communicating them and their trade-offs to others.
2. The Problem describes the criteria for when to apply the pattern. Occasionally, the problem will include a list of conditions that must be met before it makes sense to apply the pattern.
3. The Solution contains the elements that make up the design, their relationships, responsibilities, and collaborations. The solution is not a particular concrete design or implementation but a template that can be applied to many different situations.

4. The Consequences are the results and trade-offs of applying the pattern. These are important for evaluating design alternatives and understanding the costs and benefits of applying the pattern.

The four essential elements above are part of the description of a design pattern that includes a graphical representation and also a record of the decisions, alternatives, and trade-offs that led to it; as well as concrete examples. Vlissides et al., (1995) advocate describing a design pattern using a consistent format based on the following template:

1. Pattern Name and Classification – the name is a metaphor for the design and the classification places the pattern in a taxonomy of patterns.
2. Intent – a statement of what the pattern does, rationale, issues addressed.
3. Also Known As – aliases for the pattern.
4. Motivation – a scenario of the problem and how the design pattern solves the problem.
5. Applicability – situations where the design pattern is applicable.
6. Structure – a graphical representation of the design pattern, typically Unified Modeling Language.
7. Participants – the classes and/or objects participating in the design pattern and their responsibilities.
8. Collaborations – how participants carry out their responsibilities.
9. Consequences – addresses how the pattern supports its objectives.
10. Implementation – pitfalls, hints, or techniques that one should be made aware of before implementing the pattern.
11. Sample Code – software fragments that illustrate how one might implement the pattern.
12. Known Uses – examples of the pattern found in real systems.
13. Related Patterns – closely related design patterns, important differences, other patterns that work well with the one under consideration.

4.5 Organizing Design Patterns

There are many ways to organize design patterns depending on their granularity and level of abstraction. Vlissides et al., (1995) suggest two criteria: Purpose and Scope. Purpose reflects what a design pattern does. Scope specifies whether the pattern pertains primarily to classes or objects where class patterns deal with relationships between classes and their subclasses and object patterns deal with object relationships that may be changed at run time and are more dynamic, see Figure 6.

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Figure 6: Design Pattern Space (Vlissides et al., 1995, p. 10)

The design pattern space can be organized into design pattern relationships as well. Having multiple ways of thinking about design patterns deepens ones insight into what they do, how they compare, and when to apply them, Vlissides et al., (1995), see Figure 7.

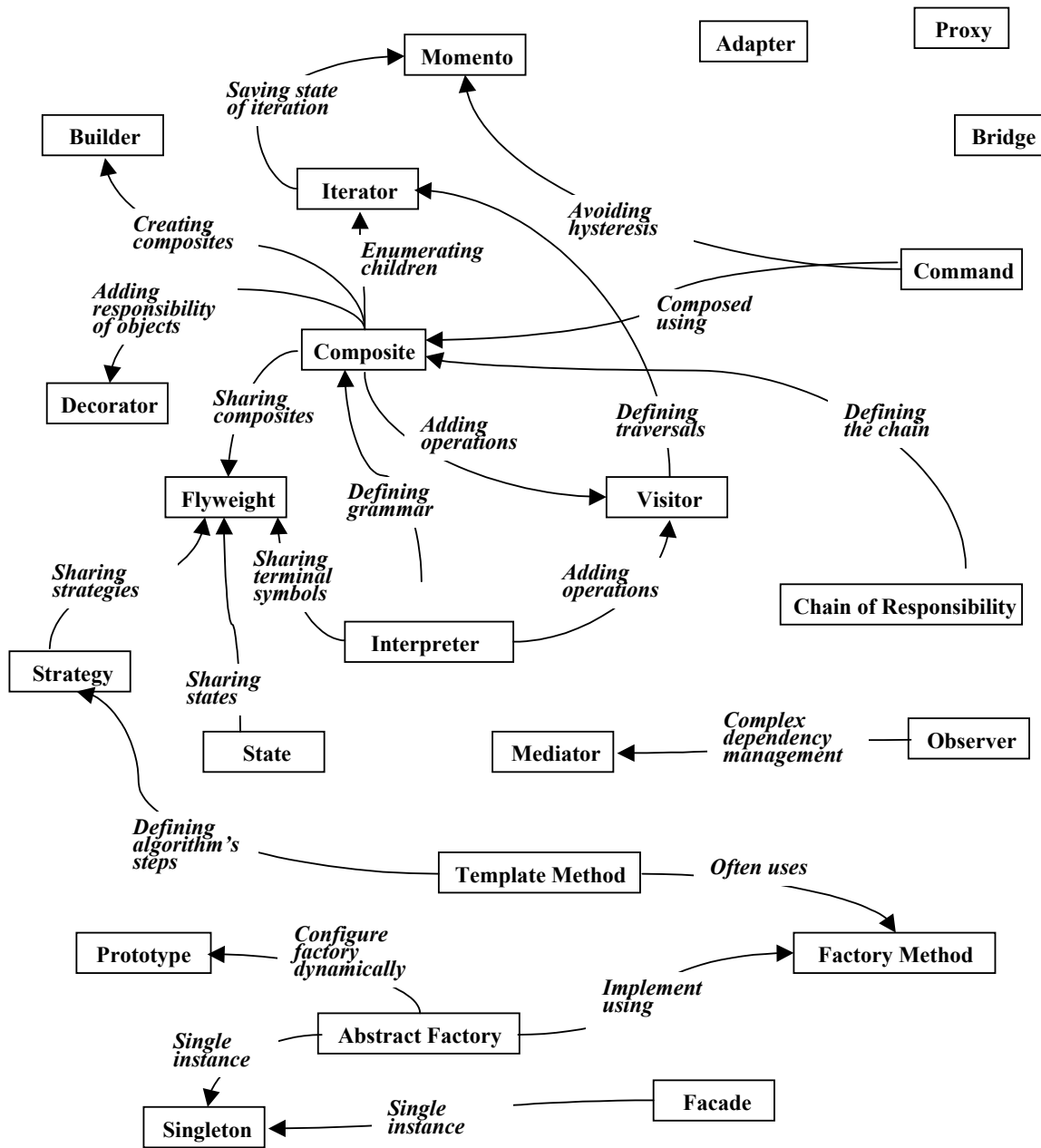


Figure 7: Design Pattern Relationships (Vlissides et al., 1995, p. 12)

Objects can vary tremendously in size and number, representing everything from the hardware or all the way up to entire applications. Design patterns address this as well (Vlissides et al., 1995):

1. Facade pattern describes how to represent complete subsystems as objects.
2. Flyweight pattern supports huge numbers of objects at the finest granularities.
3. Factory and Builder yield objects whose only responsibilities are creating other objects.
4. Visitor and Command yield objects whose responsibilities are to implement a request on another object or group of objects.

Objects interface with each other by specifying the operation's name, the objects it takes as parameters, and the operation's return value; this is referred to as the operation's signature. The set of all signatures defined by an object's operations is called the interface to the object, Vlissides et al., (1995). The object's interface characterizes the complete set of requests that can be sent to the object. Objects are known through their interfaces. Without an interface, there is no way to know anything about an object or to have it do anything without going through its interface.

4.6 Universal Modeling Language

The Universal Modeling Language (UML) is discussed relative to object-oriented software development. Fowler and Scott (1997) state that UML is a modeling language, not a method. UML has no notion of process, which is a key part of a method. There are several key diagrams within UML. For brevity, only the Class Diagram will be discussed; the set of UML diagrams and their purpose are summarized in Figure 8 (Fowler & Scott, 1997):

Technique	Purpose
<i>Activity Diagram</i>	Shows behavior with control structure and encourages parallel behavior.
<i>Class Diagram</i>	Shows static structure of concepts, types and classes.
<i>Deployment Diagram</i>	Shows physical layout of components on hardware nodes.
<i>Interaction Diagram</i>	Shows how several objects collaborate in a single use case.
<i>Package Diagram</i>	Shows groups of classes and dependencies among them.
<i>State Diagram</i>	Shows how a single object behaves across many use cases.
<i>Use Case</i>	Elicits requirements from users in meaningful chunks.

Figure 8: UML Diagrams (Fowler & Scott, 1997)

The Class Diagram is central within the object-oriented methodology. According to Fowler and Scott (1997), a class diagram describes the types of objects in the system and the various kinds of static relationships that exist among them:

1. Associations – for example, a customer may rent a number of videos.
2. Subtypes – a nurse is a kind of person.

Class diagrams also show object attributes and their operations as well as constraints regarding object connectivity.

There are three perspectives that can be used with regard to Class diagrams (Fowler & Scott, 1997):

1. Conceptual – The diagram represents a concept in the domain under study. A conceptual diagram is drawn with little regard for the software that might implement it and is considered language independent.
2. Specification – This perspective is software based but focused on the software interfaces, not the implementation. The emphasis is types as opposed to classes.

3. Implementation – This is the software view where the design is laid bare.

Perspective is crucial to both drawing and reading class diagrams. Knowledge of the perspective is essential to interpreting the class diagram correctly.

An example of a typical class diagram is presented in Figure 9.

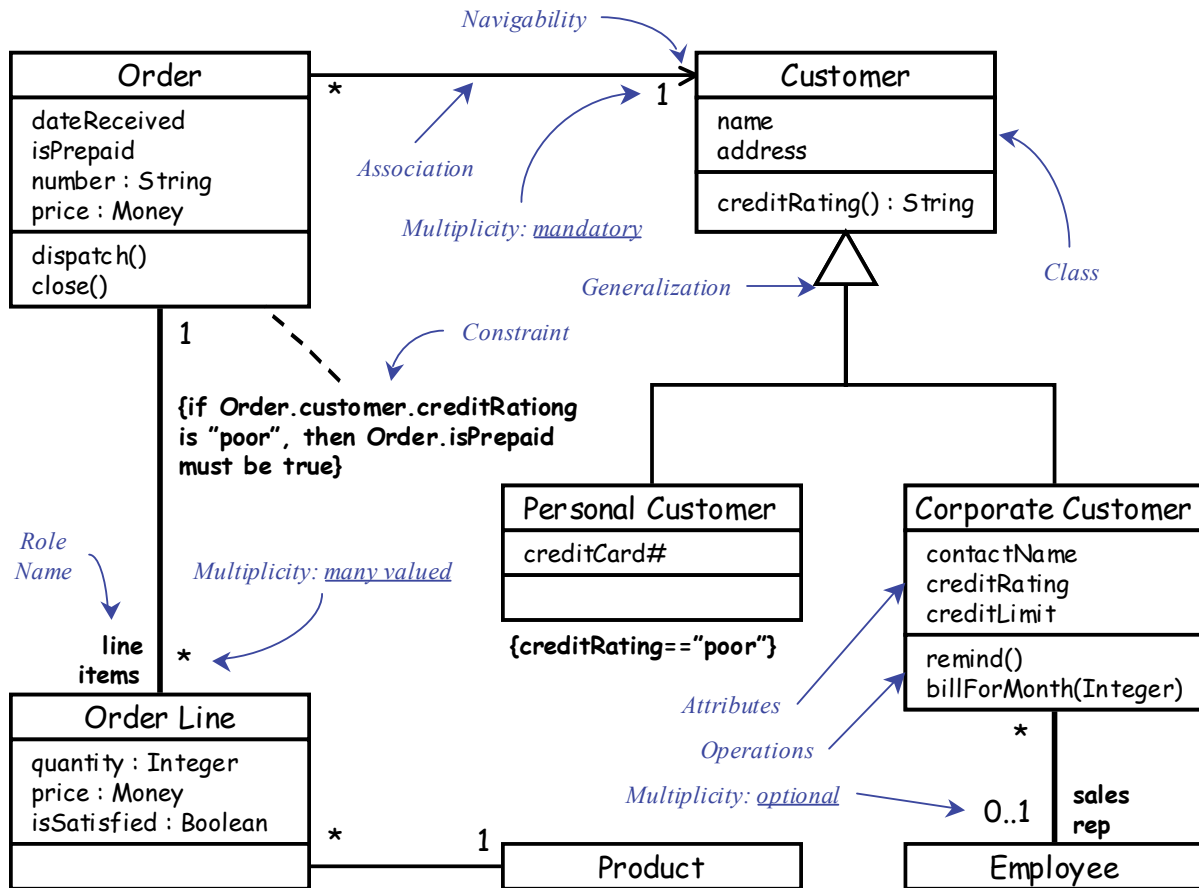


Figure 9: Class Diagrams (Fowler & Scott, 1997)

The class diagram Associations represent relationships between instances of classes, (Fowler & Scott, 1997). Associations have roles, a direction on the association. Roles can be explicitly named with a label or the name may be implied by the target class. Generally, the class that the role goes from is the source and the class that the role goes to is the target. A role may have multiplicity, an indication of how many objects may participate in the given relationship. Generally, multiplicity will indicate the range, upper and lower, for the participating objects.

From the “conceptual” perspective, associations represent conceptual relationships between classes. For the “specification” perspective, associations represent responsibilities. For the “implementation” perspective, association represents navigability. If navigability exists only in one direction, the association is unidirectional (identified by an arrow). A bi-directional association contains navigability in both directions (identified without an arrow).

Attributes are similar to associations (Fowler & Scott, 1997):

Conceptual – a customer’s name attribute merely indicates that customers have names.
Specification – a class can tell its name and has some way of setting a name.
Implementation – there is a field for the customer name.

Operations are the processes that a class carries out; they are the methods on a class.

Generalizations depend on the perspective, too (Fowler & Scott, 1997):

Conceptual – everything said at one level applies to its sublevel.
Specification – the interface of the subtype must include all elements from the supertype.
Implementation – the subclass inherits all the methods and fields of the superclass.

The class diagram itself identifies constraints. However, there is no strict syntax to capture rules as constraints in UML other than putting them inside braces ({}) and using informal English. Design patterns are described structurally using UML. UML is a near standard way of expressing design.

4.7 Applied Systems

Robinson and Kisner (1989) developed a prototype continuous simulation environment for nuclear power plants applying object-oriented programming. The system was modeled by creating a collection of objects that communicated with each other via message passing. Their workstation environment allowed them to build simulation models by selecting iconic representations of power plant components from a menu and connecting them with the aid of a mouse “click”. Using LISP as the development language, they were able to modify the models graphically at any time, including while the simulation was running, and to observe the results immediately via real-time graphics. The use of object-oriented programming allowed them to create a highly interactive and automated simulation environment.

At the core of their simulation package was a Class Library with classes grouped into four categories (Robinson & Kisner, 1989):

1. Component-level objects – the basic building blocks from which the models were made. They represented either actual components (e.g., pumps, pipes, and valves) in the nuclear plant or abstractions such as heat sources and transport delays. These objects had methods for computing their gross averaged physical properties, hydraulic, and/or heat transfer characters. The methods were localized and did not describe how the objects interacted with neighbors.
2. System-level objects – the class of system-level objects was used to define operations on groups of related objects. However, to preserve the modularity of the object-oriented approach, system parameters were referenced to the component-level objects. Thus, future changes in a component-level were immediately reflected in its associated system-level object(s).
3. Hybrid Objects – function at both the system and component levels.
4. Interface and Simulation Control – included classes to manage the simulation, manage windows and keep a file system interface.

Robinson and Kisner (1989) said that the object-oriented approach’s advantage was most significant relative to the concept of class, inheritance, and polymorphism. They found classes useful for building reusable generic descriptions of similar types of objects. This was particularly

powerful for them since their problem domain typically consisted of a large collection of basic components falling into a relatively small number of categories (e.g., tanks, pipes, and valves). These categories were conveniently described by class definitions with particular components being created through reuse by instantiating new components and overriding the default values of the previous component as necessary.

Inheritance was exploited by creating a class hierarchy where generic attributes of similar objects were defined in top-level classes inherited by lower levels. The use of a class hierarchy allowed for code to be reused and supported better maintainability (Robinson & Kisner 1989).

Polymorphism was useful since it allowed each class to define a unique response to the same message. In a traditional program, there would have to be *a priori* knowledge of the type of each component in order to determine the correct subroutine for computing a function such as “pressure drop”. This would necessitate test clauses in the code to determine the correct subroutine with a proliferation of subroutine names. With an object-oriented approach, the same message was sent to all objects in the loop “compute-pressure-drop”, leaving it to the object itself to use the correct procedure, (Robinson & Kisner 1989).

Raczynski (1990) looked at the existing software at the time and saw that much of it became easily obsolete because it was not object-oriented, regardless of whether the simulation was discrete, continuous or combined. To him simulation software had to be object-oriented because the real world being simulated is made of objects. Being object-oriented meant describing the properties of objects (e.g., behavior, interactions with other objects), and creating and handling the objects. He included defining the structure within classes of objects through the inheritance mechanism. Inheritance allowed extending the complexity of the model using classes of objects created earlier.

According to Raczynski (1990), inheritance enabled programmers to create classes and therefore objects that were specialization’s of other objects; this enabled programmers to create complex models by reusing code created and tested before. “Thus, the user can prepare and store some useful source ‘capsules’ and use them while creating new processes”, (Raczynski, 1990, p. 912).

Raczynski (1990) developed an object-oriented language based on PASCAL called PASION. He compared the elements of PASION to those commonly used in simulation language at that time. A summary of his comparison is presented in Figure 10, below (Raczynski, 1990, p. 912):

COMMON MODEL	PASION PROGRAM
Components	Objects
Component Specification	Process declaration (object type)
Descriptive variables (including the state of the component)	Process Attributes
Component activities and the rules of interaction	Events
Experimental frames	Process hierarchy and inheritance

Figure 10: Comparison of Common Model and PASION

Components were described as the elements of the simulation model (e.g., clients in a shop). The state of each component was described by the corresponding set of descriptive variables and their

activities given by the rules of interaction between the components. Experimental frames were the actual set of descriptive variables used to determine the complexity of the model.

Basnet, et al. (1990) believed that object-oriented programming offered the potential to be a major contributor to the continuing growth of simulation and modeling, and the construction of models and modification of existing models. They said that the principal idea associated with object-oriented design was that all system items (e.g., variables) are treated as “objects”, Basnet, et al. (1990). To them an object was a class or an instance of a class, where a class was the software module that provided a complete definition of the capabilities of members of the class. The capabilities were provided by the procedures and data storage contained within the immediate class definition, or inherited from other class definitions to which the class at hand was related.

Basnet, et al. (1990) cited four key object-oriented concepts that made models more understandable, modifiable, and reusable:

Encapsulation – an object’s data and procedures were enclosed within a tight boundary that could not be penetrated by another object.

Messages passing – as a result of encapsulation, messages were the means of communication from one object to affect the internal condition of another object.

Late binding – provided by object-oriented design delayed the process in which a procedure and the data on which it operated were related until the software was actually running as opposed to the time of code construction in traditional procedural languages.

Inheritance – provided for a low-level form of software reuse where object-oriented classes were defined in a hierarchical tree structure.

The concepts underlying object-oriented software were extendable to simulation modeling (Basnet, et al., 1990). They said that in terms of the simulation modeling requirements, following the object-oriented approach preserved the bulk of the developed code for general use in model building. Each model building exercise performed the particular functions that were of interest at the time. The object definitions remained independent of the functions of the system being modeled. The characteristics of the object-oriented approach allowed the rethinking of the entire approach to systems modeling using computers, (Basnet, et al., 1990).

Basnet, et al., (1990) created an object-oriented modeling environment for manufacturing systems, Figure 11. The distinctive attribute of their environment was the modular representation of physical and information/decision components; they provided a set of distinctive formalisms to support this separation. Other attributes of the modeling environment included the high-level model specification language, the construction of a library of simulation objects, and the provision of a graphical user interface.

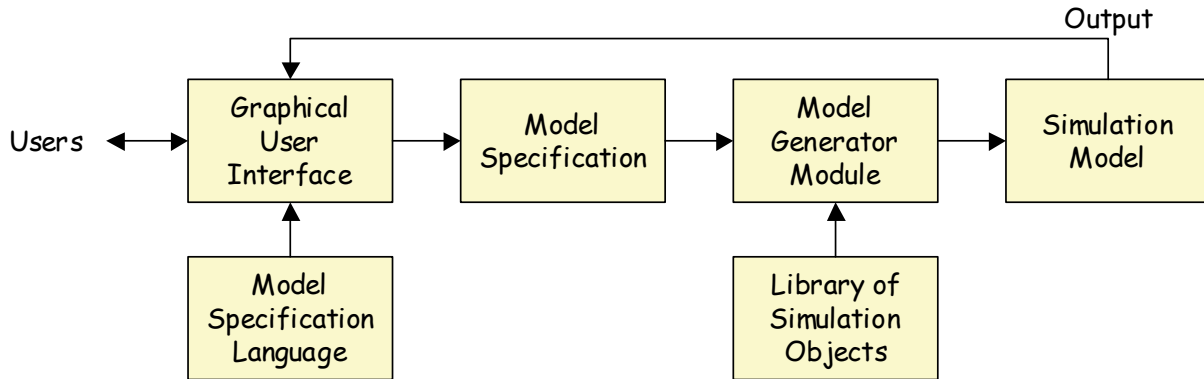


Figure 11: Manufacturing Systems Modeling Environment Architecture

According to Basnet, et al., (1990), manufacturing systems were highly influenced by control policies used in its operations. Therefore, in evaluating system performance, it was necessary to consider the physical components as well as the policies. They said that in manufacturing systems there was a complex hierarchical decision making structure; and the decisions were based on available information that was often incomplete, inaccurate or delayed. So, the decision-maker at each level of the hierarchy used heuristics, personal experience, company rules, and policies to reach control decisions. Traditional modeling tools did not provide convenient structures for specifying these decisions, (Basnet, et al., 1990); consequently, in simulation modeling, the representation of controlling influences was often embedded into elements of code modeling physical components.

Basnet, et al., (1990), cite the following reasons why a modeler would want to incorporated explicit and separate information processing and decision making structures into their model:

1. To obtain a more realistic model of the system, and
2. To determine the effect a certain operating policy will have on the system performance.

In terms of manufacturing systems, simulation languages failed in this regard, Basnet, et al., (1990); they did not provide realistic constructs for modeling information flows and control decisions. Additionally, the constructs they provided had to be hard-coded, and dispersed into the model, creating code that was hard to modify (Basnet, et al., 1990). They suggested that a new paradigm was needed to capture the dynamics of information processing and decision making as well as the manufacturing physical processes.

Basnet, et al., (1990) proposed a model specification language to capture the fundamental structure and behavior of the system elements. Their current manufacturing modeling system required the “translation” of the physical, information and decision components into the proposed high level language. They based their model specification language on Smalltalk.

The Smalltalk simulation objects were classified into two broad categories, (Basnet, et al., 1990):

1. Objects providing the software functions which allowed the background simulation processing tasks to be performed (e.g., time advance, event triggering, entity creation, list processing)
2. Objects providing the reusable building blocks for modeling manufacturing systems (e.g., machines, material handling vehicles, conveyors, work orders, routings).

Basnet, et al., (1990) pointed out that as the capabilities of the manufacturing system model were enhanced, the inheritance capability of the object-oriented approach created subclasses of the generic objects that more completely modeled the behavior of specific items. The building blocks employed a higher level of abstraction than the currently available simulation languages of his time.

Bishak and Roberts (1991) stated that the appeal of an object-oriented approach to simulation was attributable to the fact that the world consists of “objects”. When modeling a hospital floor, there were lots of objects – doctors, nurses, examining rooms, medical records, x-ray machines. Likewise, it was natural to describe things that were not physical, as objects – a database record; the symbol y may be an object that represents a variable.

Bishak and Roberts (1991) pointed out the following areas of special potential for object-oriented simulation:

1. Graphical representation of objects had the potential for animation of the objects when the simulation executes.
2. Combining artificial intelligence with objects presented the opportunity to exhibit “learning and adaptability” through encapsulation.
3. Because of encapsulation, there was the potential for the parallel execution objects and the subsequent acceleration of the speed of the simulation.
4. Lastly, objects present the opportunity for users to build their own simulation elements; this gave rise to the notion of simulation software engineering.

However, despite the potential power of the object-oriented approach to simulation, Bishak and Roberts (1991) also identified some potential problem areas:

1. The object-oriented approach represented a major paradigm shift from the “usual” procedural orientation and typically, the ability to change basic object representation remained with the software house that created the original objects.
2. To capitalize on the object-oriented approach will require the user to become somewhat of a language designer. In addition to the predefined objects, tools may be employed for crafting one’s own objects.
3. The object-oriented approach demanded that everything be represented as objects that may be difficult to grasp (e.g., thinking of a queue as an object containing objects or a server as an object servicing objects in a queue).
4. Related to the creation of objects by the user was the management of the objects. When objects are destroyed some references to them must be destroyed. Similarly, when references are destroyed so must the object be destroyed; this is sometimes referred to as “garbage collection”, not something the typical user will want to do.
5. Lastly, they suggested that dynamic binding can be a curse as well as a benefit in that late binding may slow execution and that this condition may be exacerbated by message passing which is key to the object-oriented approach. They contended that late binding placed additional responsibility on run-time software to identify the appropriate properties (variables and functions) to be obtained.

Corbin (1994) described a development technique for model conceptualization integrating archetypes and their corresponding generic models into a framework. He stated that model conceptualization was the most difficult stage of the modeling process and the most difficult to master. To conceptualize a model, the following was needed, see Figure 12 (Corbin, 1994):

1. The basic feedback structure
2. The level of aggregation
3. The model boundaries, and
4. The timeframe.

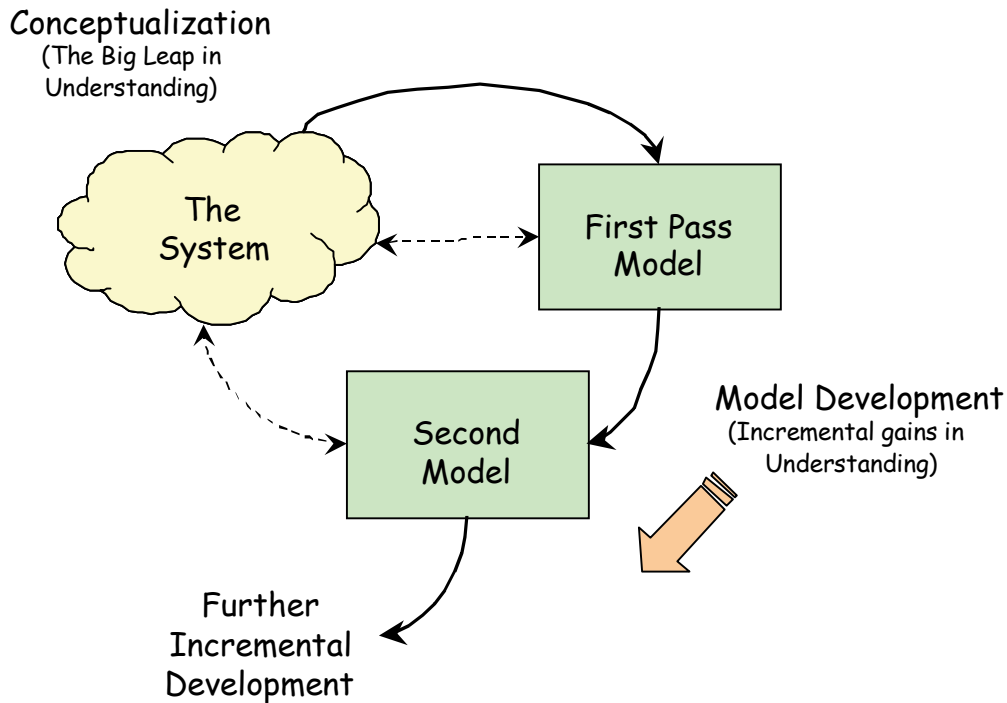


Figure 12: Conceptualization Modeling Process

Corbin (1994) identified three generic structures, see Figure 13, for further classification of their content:

Generic Models	Structures generic to a specific problem domain.
Archetypes	Structures transferable between different problem domains.
Building Blocks	Sub-structures found as building blocks in many different models.

Figure 13: Three Generic Model Structures

Corbin's (1994) conceptualization model used the "base" archetypes and generic models of those archetypes to transition to a simple working model. The "base" archetypes, see Figure 14, were based on the work of Wolstenholme and Corbin (1993):

		Intended Action	
		Control	Growth
System Reaction	Opposition	B/R Fixes that Fail	R/B Limits to Success
	Competition	B/B Fighting for Control	R/R Success to the Successful

Figure 14: Base Archetypes

Using the base archetypes, Corbin (1994) suggested basic steps for the conceptualization process as follow:

1. Specify the intended Behavior – Is the aim growth or control the intended behavior loop.
2. Identify the System Reaction – Will the system respond with growth or control.
3. Create the Base Archetype – Link the loops identified in 1 & 2 to create a base archetype.
4. Specify the Problem as a Generic Model – Take the simulation language specific model corresponding to the base archetype and customize it to represent the domain problem
5. Qualitative First Pass Model – Flesh out the loops with intermediate variables and organizational boundaries
6. Quantitative First Pass Model – Add extra detail to the model to keep it consistent with the qualitative model
7. Iterative model development – Develop the model from here on based on iterative simulation results.

One caution offered by Corbin (1994) when using this methodology based on archetypes was to strike a balance between the initial structure from the archetype and the danger of using an overly prescriptive structure that constricted the developers thinking about the problem, i.e., forcing the problem to fit the solution. In fact, Corbin (1994) felt that building the proposed framework around the full set of system archetypes would be too restrictive with the archetype not only being the starting point but also the ending point!

La Roche (1994) identified the concept of a template for the structure of a system dynamics model realized in the MicroWorld® software of DYNAMO PD+®. The Template Simulator was organized into four segments (La Roche, 1994):

1. MicroWorld
2. Infosystem
3. Controls
4. Coupling of process-chain and accounting.

The “Template-loops”, La Roche (1994), comprised a very simplified structure of a business with subsystems that defined its behavior and profit:

1. A supplier with his own planning
2. A production process-chain
3. A production and supply control system
4. A sales operation trying to match backlog and market-driven delivery delay allowed.

La Roche (1994) discussed the concept of using scenarios for business-process-engineering as general tasks to maximize asset-turnover and net product contribution, depending on the type of demand variation the business was subject to. He identified fundamental types of process-chain control with application to the basic model (La Roche, 1994) as illustrated in Figure 15:

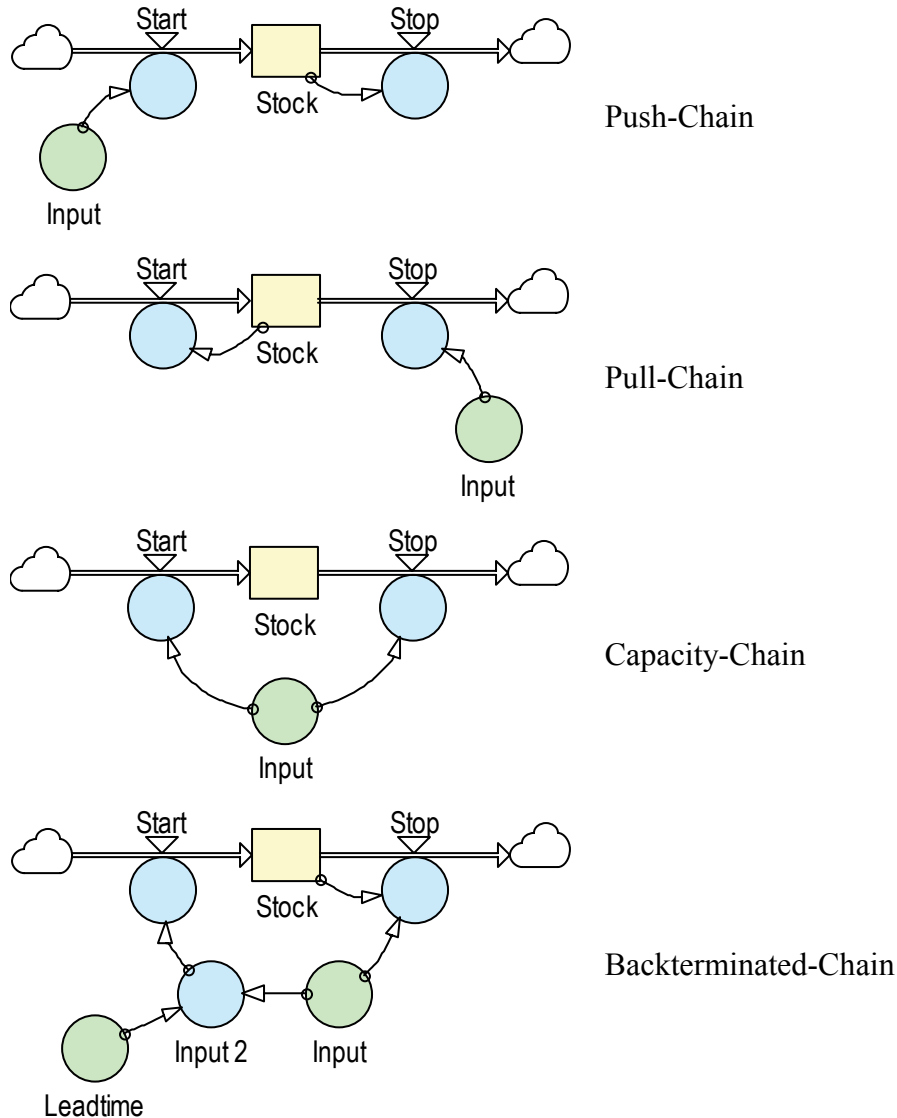


Figure 15: Process-chain control structures

La Roche (1994) said that using a template model at the start of the modeling process built on the essence of broad experience in the field. Templates lent themselves to an interactive and repetitive model building process (La Roche, 1994). Model building started with a provisional problem exposure of the people concerned with business process engineering using the template model and adjusting its parameters to fit the case at hand:

1. Expanding the template model structure towards the actual business process-chains.
2. Putting the pre-tested subsystems together as an updated version of the customized business-model.

La Roche (1994) believed that a continuous top-down model of the business-process-chain would be a useful and versatile tool to get the grand picture of the really worthwhile improvement of the process.

Joines and Roberts (1994) prepared a tutorial showing how to design object-oriented simulation models using the C++ language. The conceptual design of the object-oriented context for simulation is illustrated in Figure 16 (Joines and Roberts, 1994):

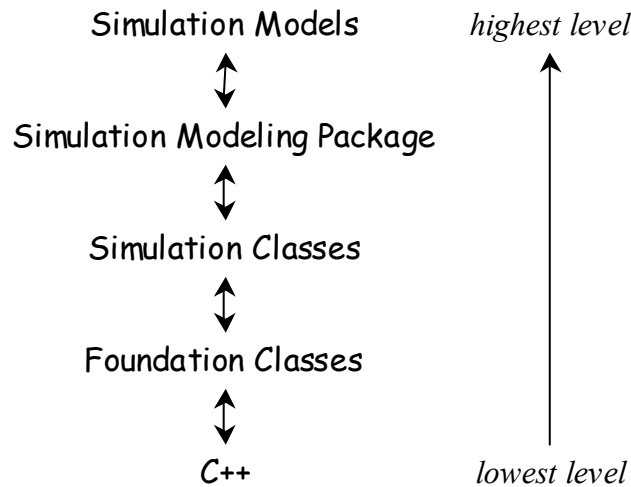


Figure 16: Context Design for Object-Oriented Simulations

To Joines and Roberts (1994) “users” could relate to the design at any level. If only interested in results, the user related at the model execution level. Or, if only interested in the algorithm construction level, they related at the C++ level. Because of the object-oriented design, the concepts at each level were “encapsulated” so users need not be concerned with the concepts at the lower level. But, the sophisticated user had access to the lowest possible level.

The class concept is fundamental to object-oriented software according to Joines and Roberts (1994); the class provides a “pattern” for creating objects and defines the “type”. An example of an *Exponential* class follows (Joines & Roberts, 1994), see Figure 17:

```

#include "random.h"
/* "expon.h" contains the class Exponential. This class describes an
   inverse transformation generator for Exponential variables. */

class Exponential : public Random {
public:
    Exponential(double, unsigned int=0, long=0);
    Exponential(int, unsigned int=0, long=0);
    virtual double sample()
    void setMu(double initMu) { mu = initMu; }
    double getMu() { return mu; }
private:
    double mu;
};
  
```

Figure 17: Class definition of object’s Properties

Joines and Roberts (1994) explained that the properties of classes, data objects and functions, were grouped into “public” and “private” sections of the C++ software. Public properties were accessible from outside the object. Private properties were accessible only from within the object and were locked-out to the public. Making a property private restricted unauthorized use and encapsulated the object’s properties. The *Exponential* class inherited from the *Random* class and had access to all the public properties of the *Random* class without having to re-code them. There were two constructors in the *Exponential* class; one takes a “double” and the other takes an “integer”. Likewise, a destructor, not used in *Exponential* class, will cleanup any object responsibilities. The *sample()* function was specified as a virtual function in *Exponential* because the type of variable was not known a priori. The program decided at run-time which random variable to sample; “run-time” binding; this makes the entire specification of sampling from variables much simpler. As an illustration of polymorphism, the *Exponential* class had two constructors so users may specify either floating point or integer arguments for the mean interarrival time. Polymorphism allowed the same properties to be applied to different objects, i.e., integer or double. Under other circumstances, polymorphism will allow users to produce the same behavior with different objects.

Myrtveit and Vavik (1994) investigated modeling as a way of learning, and learning from running simulations. They found that to meet new requirements for learning environments that concrete objects were needed in addition to the general and abstract objects of accumulator-flow diagrams. Myrtveit and Vavik (1994) found that the use of objects was an elegant way to break the “world” into smaller parts that were easier to handle. To them, classification of objects was significant. Objects with the same properties (attributes) and operations were grouped into a class, Myrtveit and Vavik (1994). An attribute or operation local to a class was hidden (encapsulated) inside the class.

Myrtveit and Vavik (1994) thought that only in rare circumstances would an accumulator-flow diagram represent a natural object mapping a system. To them accumulator-flow diagrams focused on object attributes, and relationships between attributes. This focus was natural since the main purpose of accumulator-flow diagrams was to describe the dynamic relationships between attributes of a system and deduce the resulting behavior over time. To Myrtveit and Vavik (1994), providing higher level objects may be a way to make modeling useful to non-modelers.

Senge (1994) discussed seeing patterns of structure recurring again and again; he referred to these structures as archetypes and acknowledged that they recur in many different areas of knowledge: biology, psychology, economics, political science, management and ecology. To Senge, archetypes provided hope that specialization and fractionalization of knowledge would be bridged. Archetypes, per Senge (1994), were made of system building blocks: reinforcing processes, balancing processes, and delays. The structure of a frequently recurring archetype is illustrated in Figure 18, Limits to Growth, (Senge, 1994, p. 97).

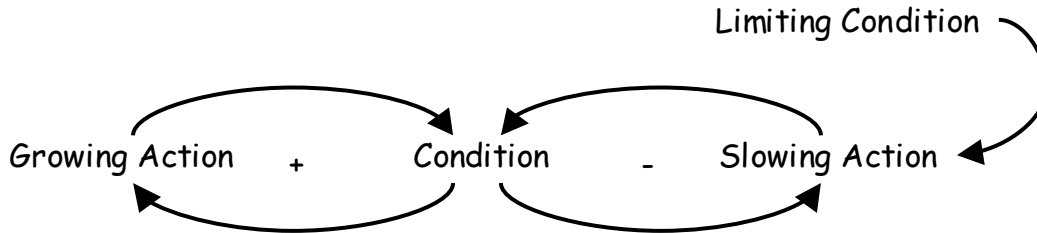


Figure 18: Limits to Growth Archetype

With the Limits to Growth archetype, a reinforcing (amplifying) process is set in motion to produce a desired result. A spiral of successes resulted; but they also created inadvertent secondary effects that eventually slowed down the success rate. According to Senge (1994, p. 95), the management principle of the Limits to Growth archetype is as follows: “Don’t push growth; remove the factors limiting growth”. Senge (1994) says that there is approximately a dozen system archetypes that affect us.

Goldgar and Acosta (1995) raised the perspective for object-oriented design from the software level to the system level for systems engineering of large, complex systems. They said that in 1995 systems were created that were two or three orders of magnitude greater in complexity than those of only five or eight years earlier were. They claimed that the more complex systems were developed with the same tools and methodologies of the early period. They claimed that the emergence of object-oriented analysis methods integrated several standard system engineering modeling paradigms, e.g., entity-relationship models, state transition models, and process or functional models.

The particular short fall of interest to Goldgar and Acosta (1995) with system engineering models regarded provisions for system performance modeling, e.g., shared resource contention, queuing, resource utilization, and response time. They took advantage of logical system definition facilities of object-oriented analysis methods based on Shlaer-Mellor. They concluded that by taking advantage of powerful object-oriented and performance modeling abstractions, a foundation was provided for a system engineering discipline that encouraged functional definition, performance evaluation, and system partitioning early in the system lifecycle. To Goldgar and Acosta (1995), a comprehensive analysis of system requirements and design reduced the risk, cost, and time involved in constructing and deploying complex computer-based systems.

Eberlein and Hines (1996) published their first iteration of “molecules” that described fundamental System Dynamic capabilities. The molecules included the following:

1. Name
2. Parents
3. Used by
4. Category
5. Problem Solved
6. Equations
7. Description
8. Behavior
9. Classic Examples
10. Caveats
11. Technical Notes.

An example of a “molecule” is provided in Figure 19 below.

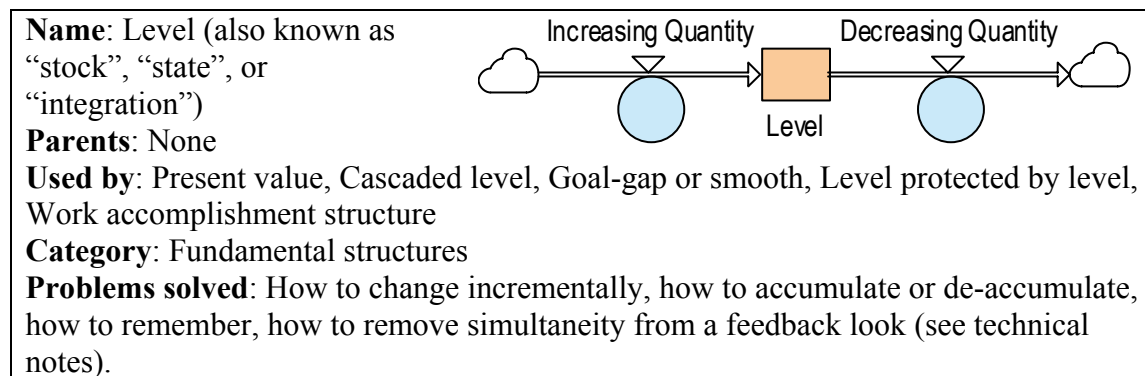


Figure 19: Molecule Illustration of a Level

Ahmed (1997) pointed out that traditional system dynamic software allowed users to build models from abstract primitives. He felt that this process was slow and required deep skills in both the discipline of modeling and the domain of the subject model. These two items, he claimed, were impediments to potential users and proposed a methodological process based on components that brought those with problem domain knowledge closer to the modeling domain without prerequisites of deep knowledge of the modeling process.

According to Ahmed (1997), component design was not new to software engineering; and with the increasing focus on object-oriented design, there was a great potential for reuse of parts or whole existing models. To this Ahmed (1997) focused on the specification of the requirements for component specification and design.

His work differentiated components from generic structures and molecules. Ahmed (1997) declared that a component was built from a collection of variables and a number of components could be configured into a model. To Ahmed (1997), components had two main features:

1. Specification, and
2. Implementation.

To Ahmed (1997), there were clear benefits to be derived from the use of components:

1. Enabling business professionals or engineers not trained as modelers to build models.
2. Shortening the development time and lowering the cost of producing models.
3. Allowing modelers to leverage each others components in their own works, and

4. Enabling model developers to concentrate on specific features of their models due to the availability of third party components.

To this end, Ahmed (1997) advocated a component catalog architecture, Figure 20.

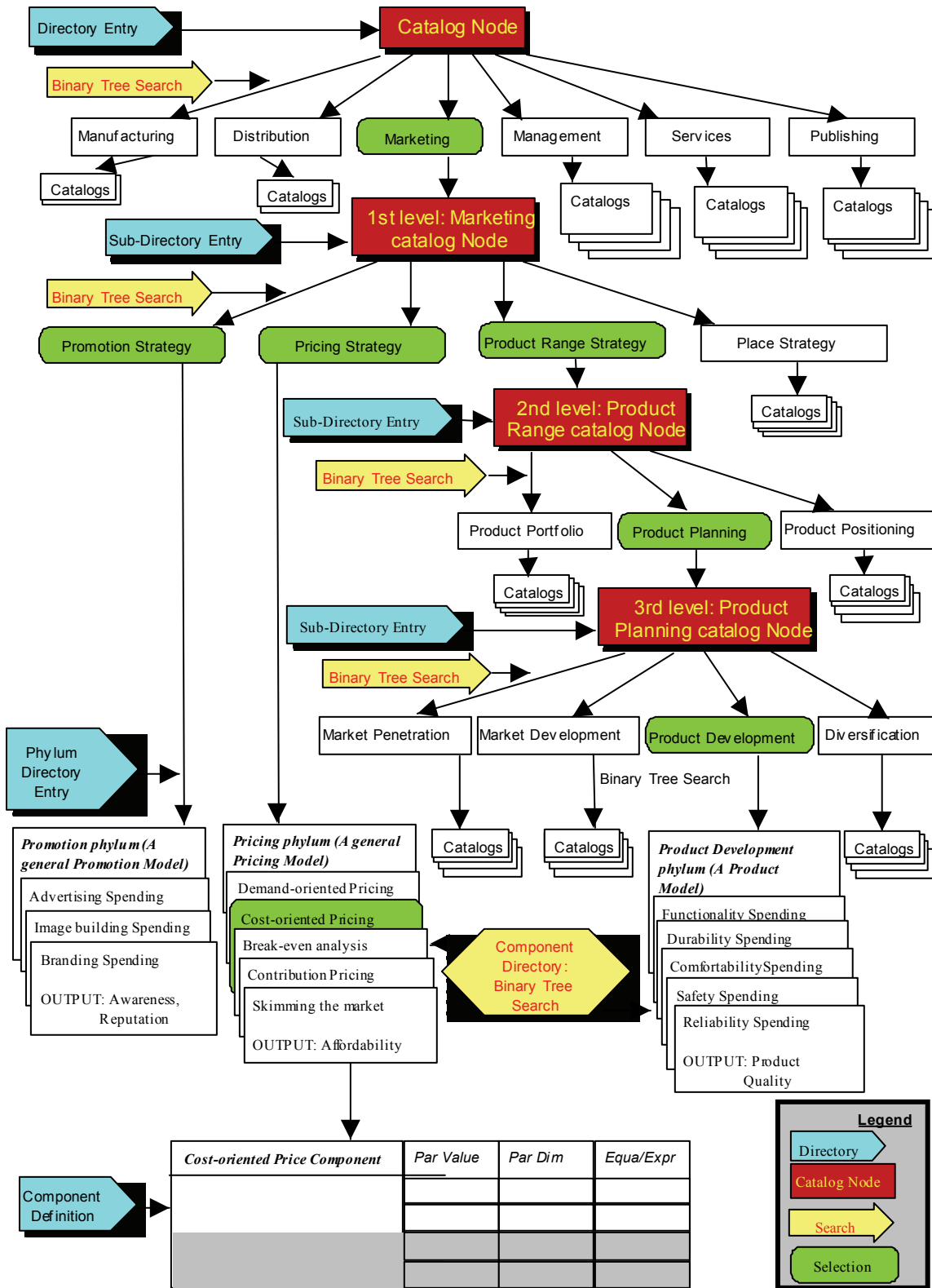


Figure 20: Component Catalog Architecture

Essentially, Ahmed (1997) said that modeling tools needed to be made available to practical business people, politicians and other professionals so that they could have an opportunity to learn to control unintuitive dynamic processes.

Kovács, Kopácsi, and Kmecs (1997) noted that software developers often experience the problem of creating components for an application that someone has produced previously. Without effective reuse tools, it is natural to create components from scratch than look for useful components in other programs or systems. In the field of flexible manufacturing systems and flexible manufacturing cells, this is often the case. Even though the components of flexible manufacturing systems and cells are the same types of machine tools, robots, and transfer equipment, the components are recreated rather than reused. Typically, they will differ from each other only in their amounts and working parameters, Kovács et al., (1997).

Kovács et al., (1997) investigated the design methodology based on the object-oriented Rational Rose CASE tool. They concentrated not only on the software reuse but the documents created during the conception, design, implementation, and testing phases.

They found that components can be analyzed and defined using these tools and reuse achieved. The reuse and application of the object-oriented design techniques helped them to build different flexible manufacturing systems simulation models easier, faster and more reliably, Kovács et al., (1997).

Kortright (1997) used the UML as a modeling and simulation language with Java as the implementation language. Based on the Model-View-Controller design pattern, he added different views to the models for statistics collection, animation, and checkpoint recording. The Model-View-Controller design pattern significantly facilitated simulation model building by disassociating a model from event handling, statistics gather, and other observable functions, Kortright (1997).

He found that he could use the same simulation model for sequential and parallel discrete-event simulations simply by exchanging controllers, without changing the model itself, Kortright (1997). Similarly, an arbitrary set of views was added to the model.

Kortright (1997) investigated UML, a third generation object-oriented modeling language, to represent simulation models. In UML, models were described through a rich set of diagrams, Kortright (1997):

- 1) Class Diagrams described various object classes and their relationships and associations, including inheritance and aggregation.
- 2) Use-case diagrams described the intended use of the model system.
- 3) Interaction Diagrams showed the timing and sequencing characteristics of the system. There were two types of diagrams used here:
 - a) The Sequence Diagram described message passing as time flows in a system and included annotations for specific timing requirements.
 - b) The Collaboration Diagram showed message passing without reference to a time axis, allowing for the description of a scenario while keeping clear the structure of the system.

- 4) State Diagrams allowed for guards on transitions, propagated transitions, actions on transitions, actions on state entry, and more. The diagrams accounted for both concurrent and hierarchical state diagrams.
- 5) The Component Diagram mapped the model to a set of implementation components.
- 6) The Deployment Diagram described the binding of software components to physical devices.

Kortright (1997) found UML comprehensive and under consideration to become an official standard in software engineering. He felt that there was a great deal to gain by using UML for simulation modeling. To him an important problem was that simulation models were described in a large variety of notations, or directly in a programming language. UML “provided a set of proven notations for model description and permitted the visualization of a number of alternative designs within an integrated framework”, (Kortright, 1997, p. 44).

The Model-View-Controller design pattern, Figure 21, kept the underlying model separate and independent of event handling and viewing mechanisms, Kortright (1997). Mechanisms developed without using the design pattern mixed the simulation model with viewing and event-handling functions, making it more difficult to modify or extend the model (Kortright, 1997).

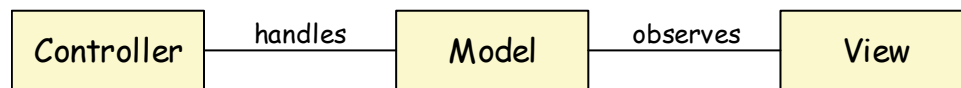


Figure 21: Model-View-Controller Design Pattern

Kortright (1997) found a close correspondence between Java and UML and observed the following:

- 1) UML classes map to Java classes.
- 2) UML operations map to Java methods.
- 3) UML interfaces map to Java interfaces.
- 4) UML inheritance relates to Java implementation through the “extends” and “implements” relationships.

Typically, UML constructs corresponded to an equivalent Java construct. Therefore, Kortright (1997) observed that Java could implement large-scale, multithreaded, distributed systems that UML described.

Savino-Vázquez and Puigjaner (1999) used UML to specify the structural components of queuing network performance models. They started with the problem of representing large-scale systems with multiple components in a simulation model in order to facilitate reuse, maintenance and testing.

Robinson and Whisenhunt (1999) applied an object-oriented approach to model a PowerPC with the POWERSIM language. MOOSE (Motorola Object-Oriented Simulation Environment) provided a distributed object-oriented simulation kernel. Each object in the simulation corresponded to a specific hardware component with practically a one-to-one correspondence between simulation objects and distinct hardware components. The simulation objects were configurable to model a series of computers from reuse of basic subsets of software “parts”, i.e., objects, Robinson and

Whisenhunt (1999). The design of MOOSE emphasized encapsulation of models and ease of use.

Myrtveit (2000) defines object-oriented extensions to the basic SD language of levels and flows. Basic SD has only built-in classes, called levels and auxiliaries. Myrtveit introduces user-defined classes, called components. The SD counterpart of an object is a variable. The submodel variable type is introduced to create hierarchical SD models, and to instantiate components. Links and flows are the SD counterparts of object relationships. Myrtveit extends this to wire connections, where parameters are bundled together into type-safe connections between variables.

5 Brief Description of the Research Method and Design

The research method compared the object-oriented criteria as defined by Taylor (1990), design patterns by Vlissides (1995), and the UML by Fowler and Scott (1997) to the literature research results. The literature research articles were examined for use of and reference to use of object-oriented design, design patterns and UML.

6 Data Analysis

Review of the literature resulted in clusters of articles around the following categories:

- 1) Structure and Design Patterns
- 2) Object-oriented Design
- 3) Unified Modeling Language.

The articles that occupy these categories dated from 1989 to the present with the exception of the UML category. Articles that discussed simulation and modeling using UML were very recent and few in numbers, only three articles found, Figure 22.

Author(s)	Structure, Design Patterns	Object-oriented	Unified Modeling Language
Robinson and Kisner (1989)		.	
Raczynski (1990)	.	.	
Basnet, et al. (1990)		.	
Bishak and Roberts (1991)	.	.	
Corbin (1994)	.		
La Roche (1994)	.		
Joines and Roberts (1994)	.	.	
Myrtveit and Vavik (1994)	.	.	
Senge (1994)	.		
Goldgar and Acosta (1995)		.	
Eberlein and Hines (1996)	.		
Ahmed (1997)	.	.	
Kovács, Kopácsi, and Kmecs (1997)	.	.	.
Kortright (1997)	.	.	.
Savino-Vázquez and Puigjaner (1999)		.	.
Robinson and Whisenhunt (1999)		.	
Myrtveit (2000)	.	.	

Figure 22: Summarizing Articles Researched and Categorization

Analysis showed that simulation modelers prolifically generated terminology to describe their craft. To focus the paper, the term component (s. Myrtveit 2000) was used for a model “class” that can serve as a building block when creating model “objects”. Components have interfaces defining the variables that carry information between the components and the rest of the model. Design patterns can be used both to implement and to document components.

Analysis of the literature survey terminology is essential to understanding object-oriented design patterns and system dynamic components. A brief discussion of terminology follows:

- 1) A System Dynamics *model* consists of variables.
- 2) Basic System Dynamics uses *basic variables*, which are state variables and non-state variables. There are many synonyms for the basic variable types. State variables are called stocks, levels, accumulators, and reservoirs. Non-state variables are called auxiliaries, converters, and constants.
- 3) Extended System Dynamics (Myrtveit 2000) allows variables to contain other variables to an arbitrary level of nesting. All the variables of a model reside within one top-level variable, called the *model* variable. A *component* is defined as a model and its visualizations (diagrams). Components can be used as *submodels* of other components. Components correspond to classes. Submodels correspond to objects.
- 4) Some variable types, e.g., *queue* can to some extent be consider built-in structured variables. These components are not user-defined, and cannot be modified or inspected (black box).

- 5) Structured variables (Myrtveit 2000) have a type. The type determines possible connections between variables. A *socket* can be connected to a *plug* of the same type, and components of the same type can be freely exchanged. This mechanism opens up for *polymorphism*.
- 6) *Polymorphism* implies that one component can be exchanged for another component without the need for the using component to know. As an example, if a component S of type Supplier is connected to another component P, then S can be replaced by any other component of type Supplier, with no need to change the connected component P.
- 7) A *design pattern* (in System Dynamics) is a network of (structured) variables, connected together to solve a specific problem.
- 8) A *molecule* can be considered a design pattern that uses only basic variables. Molecules do not support polymorphism, since basic variables do not have complete interface definitions (The user has to fulfill the connection by manual editing of equations. An exception is flows connected to levels.). Molecules are runnable models, but they are not *classes*, as they do not implement *types*. This means that molecules cannot be reused except through copying and editing.
- 9) A class (in System Dynamics) is a component (model) that is equipped with a type (interface). The type is identified by a type name. The interface contains parameter variables for import and export of information with a model of that type.
- 10) Archetypes are "template" influence diagrams used to describe the feedback loops involved in explaining a common problem situation. Feedback loops are best described using basic variables only, and in the form of Causal Loop Diagrams. Archetypes are not runnable, in the sense that they do not specify the equations involved in the relationships between the variables.

A summary of the terminology is presented in Figure 23 below.

	Building block	Type	Composed from	Instantiable	Runnable
Object	<i>Level</i>	fixed	nothing	yes	no ¹
	<i>Non-level</i>				
	<i>Oven</i>				
	<i>Conveyor</i>				
	<i>Queue</i>				
	<i>Molecule</i>	no	basic variables	no ²	yes
	<i>Submodel</i>	user-defined	variables	yes	yes
Pattern	<i>Archetype</i>	no	abstract variables ²	no	no
	<i>Molecule</i>	no	basic variables	no	yes
	<i>Component</i>	user-defined	variables	yes	yes

¹ User must edit equation to complete definition.

² User can copy and paste molecules, but they must be edited in order to connect to the rest of the model. (Molecules do not have interfaces for connecting them up.)

³ An archetype does not even determine if a variable has a state (level) or not (non-level).

Figure 23: Summary of Terminology

Data analysis showed that "structure" is essential to simulation system design: Bruner (1960), Forrester (1990), Alexander et al., (1977), and Vlissides (1995). Within structure, the content of

simulation system design may be expressed in several ways from domain to problem specific software code. Analysis of the content of an object-oriented design as specified by Taylor (1990) revealed that continuous, System Dynamics, models fell short in the “messages” category; otherwise, the major criteria for object-oriented design applied, see Figure 24.

System Dynamics	Object-Oriented
Bounded (interfaces)	Bounded (interfaces)
<i>(not identified in literature)</i>	Messages
Variables	Objects
Levels and Rates (built-in, generic) Ovens, Queues, etc. (built-in, special) Components (user-defined)	Classes
Components Generic Structures Molecules (models) Archetypes	Patterns

Figure 24: Alignment Analysis of System Dynamics and Object-Oriented Structures

Design patterns captured the fundamental design structures and cataloged them for reuse as the basis of new simulations or the generation of additional design patterns. There was a general alignment between software engineering design patterns and system dynamic design patterns, with the exception of “messages”. Design patterns are essential to simulation model reuse.

The UML standard defines nine different kinds of diagrams, some with its own subtypes. The diagrams can be grouped static and dynamic views of systems (or models). The table below is an attempt to summarize how the various UML diagrams relate to system dynamics modeling.

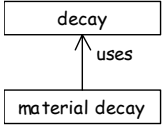
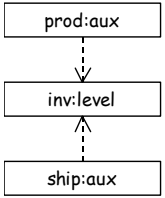
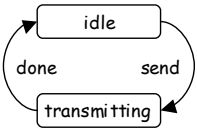
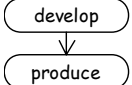
Structural Diagrams	<p>Class Diagrams</p> 	<p>Basic SD has only built-in classes. Therefore, from a class point of view, all <i>basic</i> SD models look the same. Class Diagrams, can be used, however to describe the architecture of component catalogs that contain user-defined SD classes and their interfaces. As an example, the diagram in <i>Figure 20: Component Catalog Architecture</i> can be mapped to an UML class diagram. The benefit of UML is that it is a standard for describing classes and relationships from high-level designs to low-level of detail.</p>
	<p>Object Diagrams</p> 	<p>Any SD model can be mapped to an UML object diagram. SD variables correspond to OO objects, and variable dependencies correspond to UML dependencies (dotted line with arrow).</p> <p>SD uses many different kinds of object diagrams, with vendor and author specific variations. The accumulator-flow diagrams of SD have symbols for variables and links and flows for relationships. The class of the variable is captured by the shape of the variable symbol (rectangular, circular, etc.) and the nature of the relationship by the look of the link (double line, single line, dotted line, etc.). This makes AFD a very compact object representation of an SD model.</p> <p>A SD causal-loop diagram is a simpler SD object diagram type. Here all variables are represented using one symbol (the variable name). Variable type (class) is omitted from the CLD view of a model. Object relationships are visualized using arrows. The polarity marks (+/- or s/o) that are used on links, match in a vague way to UML named dependencies.</p> <p>The diagram type used by Senge (1994) to represent archetypes is a simplification of CLD, in that link polarities are not shown. Again, there is a close map to UML object diagrams.</p>
	<p>Component Diagrams</p>	<p>Component diagrams can be used to describe the organization of model libraries, contents, interfaces and relationships between libraries. Along with the emerging OO extensions to SD, this diagram type can be a useful way to document large projects and re-usable component catalogs.</p>
Behavioral Diagrams	<p>Use Case Diagrams</p>	<p>This diagram type groups objects into actors, use cases and various kinds of relationships among them. The diagram type can be used to describe how various parts of a system interact. A use case diagram, as an example, shows how a set of components for modeling a market place for products and services can be put together to model a given scenario (e.g., two competitors, one product, two market segments).</p>
	<p>Interaction Diagrams</p>	<p>This diagram category is subdivided into Sequence Diagrams and Collaboration Diagrams. Forrester's (1990) illustration in <i>Figure 5: System Dynamic Time Sequence</i> is a kind of tabular sequence diagram. The interaction diagrams defined by UML can be used to describe the steps that take place during a simulation process.</p>
	<p>Statechart Diagrams</p> 	<p>Such diagrams are best used for displaying how objects can enter and leave states in response to certain events. In continuous models, state transitions are often modeled using flows, and the states are represented as levels (counting the objects in each state).</p>
	<p>Activity Diagrams</p> 	<p>This UML diagram type is used to model workflow and operations. The SD counterpart is the chain of flows between levels in an accumulator-flow diagram.</p>

Figure 25: Relating UML to System Dynamics

Annotations can be used to include comments into any UML diagram. This is a standard feature that can be considered also for SD diagramming languages.

7 Major Findings and Their Significance

The major findings of this research and their significance is presented as follows:

1. Object-oriented analysis and design provides a bridge between the software engineering and simulation and modeling community based on fact that simulation models are made of software. Both discrete and continuous simulation modelers, as well as software engineers strive to produce software i.e., models and applications, efficiently with as much reuse as possible. This means that modelers and software engineering have a lot of common knowledge to share and leverage in their respective disciplines, although the weight of the benefit appears to be for the modeler.
2. Design patterns as components support model reuse and may act as a basis for the design and development of many new models. This is significant from a reuse perspective. For new modelers or the uninitiated in modeling, seeing current models relevant to their own problems may enhance interest and commitment to the modeling process.
3. Terminology is prolific with many overlaps between the object-oriented and modeling communities. Note that molecule and component can be considered as objects and as patterns. In the object view, the building block is used together with other building blocks to create a model. Here instantiation is important. In the pattern view, the building block defines a network of objects that are connected up to serve a given purpose (solve a given problem). The difference here is that the problem is solved through interacting objects, rather than by one single object. Without a better understanding of the terminology and the convergence to a common understanding, the ability to leverage the knowledge of each community will be lost. UML offers an opportunity to share a common language to describe models.
4. The foundations of System Dynamics align well with the object-oriented paradigm with the exception of “messages”. However, with the notion of the time-step and the variables that it effects, the concept of message exists in System Dynamics; this is an important concept to complete the relationship between System Dynamics and Object-Oriented foundations.
5. Discrete simulation is ahead of the System Dynamics community in applying and experimenting with object-oriented design concepts. This is significant in that their reaction to using object-oriented design and experimenting with it appears positive. System Dynamics can learn from their experiences.
6. UML has just begun to be used to describe simulation models. This offers System Dynamics an opportunity to start at the beginning of a potentially significant movement towards a common software engineering design language with the potential to generate simulation code in multiple languages from C++, JAVA, to someday, VENSIM, POWERSIM and others.
7. In object-oriented design, the “class” is a fundamental concept. The literature shows that the concept of class is not new to System Dynamics but has many different names and definitions from component, to molecule to template. The significance is that the concept is converging and to reach convergence better definition of terminology is required.
8. As simulation models increase in size and complexity, the use of object-oriented design will be inevitable as an engineering discipline to manage development. The good news is that there is a discipline available to leverage, object-oriented design, if System Dynamics wants to use it.

9. Object-oriented design crosses different areas of knowledge (e.g., biology, psychology, ecology, and engineering). This is significant as a communication tool for modelers.
10. Users and customers will relate better to “objects” than other abstractions of their domains. The better “buy-in” from the users and customers the more successful simulation and modeling will be.

8 Conclusions

The major conclusions reached as a result of this research are as follow:

- 1) The work of Vlissides et al. (1995), Alexander et al. (1977), and Forrester (1990) affirm that generic patterns are a basis for problem solving.
- 2) Similarly, Vlissides et al. (1995), Bruner. (1960), and Forrester (1990) affirm that structure is fundamental to problem comprehension and understanding.
- 3) The literature shows a convergence between Object-Oriented Analysis and Design and Simulation and Modeling with the discrete modeling community ahead in this trend as compared to the continuous modeling community.
- 4) System Dynamics models, in general, meet most of the criteria to be object-oriented with the exception of message handling unless Delta Time, time-step, qualifies as an notion of “message”.
- 5) UML is just emerging as a tool in the Simulation and Modeling communities.
- 6) Design patterns can be useful in documenting and developing re-usable generic solutions on top of the emerging object-oriented extensions to System Dynamics.

9 References

- Ahmed, U. (1997). A process for designing and modeling with components. Proceedings of the 15th International System Dynamics Society. Turkey: System Dynamics Society.
- Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., and Angel, S., (1977). A pattern language. New York: Oxford University Press.
- Basnet, C., Farrington, P., Pratt, D., Kamath, M., Karacal, S., Beaumariage, T. (1990). Experiences in developing an object-oriented modeling environment for manufacturing systems. Proceedings of the 1990 Winter Simulation Conference. IEEE.
- Bishak, D. & Roberts, S. (1991). Object-oriented simulation. Proceedings of the 1991 Winter Simulation Conference. IEEE.
- Braude, E. (1998). Towards a standard class framework for discrete event simulation. Proceedings of 31st Annual Simulation Symposium.
- Bruner, J. (1960). The process of education. Boston: Harvard University Press.
- Corbin, D. (1994). Integrating archetypes and generic models into a framework for model conceptualism. Proceedings of the International Systems Dynamics Society. Sterling: System Dynamics Society.

- Eberlein, R. & Hines J. (1996). Molecules for modelers. Proceedings of the International System Dynamics Society. Cambridge: System Dynamics Society.
- Fayad, M. & Schmidt, D. (1997). Object-oriented application frameworks. Communication of the ACM40.
- Forrester, J. (1990). Principles of systems. Portland: Productivity Press.
- Fowler, M. & Scott, K. (1997). UML distilled: Applying the standard object modeling language. Reading: Addison-Wesley.
- Goldgar, R. & Acosta, R. (1995). Integration of object-oriented analysis and performance simulation for engineering computer-based systems. IEEE.
- Joines, J. & Roberts, S. (1994). Design of object-oriented simulations in C++. Proceedings of the 1994 Winter Simulation Conference. IEEE.
- Kortright, E. (1997). Modeling and simulation with UML and JAVA. IEEE.
- Kovács, G., Kopácsi, S., & Kmecs, I. (1997). Simulation of FMS with the application of reuse and object-oriented technology. Proceedings of the 1997 IEEE International Conference on Robotics and Automation. Albuquerque: IEEE.
- La Roche, U. (1994). A basic business loop as starting template for customized business-process-engineering models. Proceedings of the International Systems Dynamics Society. Sterling: System Dynamics Society.
- Myrtveit, M., & Vavik, L. (1995). Object based dynamic modeling. Proceedings of the International Systems Dynamics Society. Tokyo: System Dynamics Society.
- Myrtveit, M. (2000). Object-oriented Extensions to System Dynamics. Proceedings of the International Systems Dynamics Society. Bergen: System Dynamics Society.
- Raczynski (1990). Pasion: object-oriented simulation on the PC. . Proceedings of the International Systems Dynamics Society. Chestnut Hill: System Dynamics Society.
- Robinson, J. & Kisner, R. (1989). An intelligent dynamic simulation environment: an object-oriented approach. Proceedings of International Symposium on Intelligent Control. IEEE.
- Robinson and Whisenhunt (1999). A powerPC platform full system simulation – from the MOOSE up. International Performance, Computing, and Communications Conference. Phoenix: IEEE.
- Savino-Vázquez, N. & Puigjaner, R. (1999). UML-based method to specify the structural component of simulation-based queuing network performance models. Proceedings 32nd Annual Simulation Conference. Los Alamitos. IEEE Computer Society.
- Senge, P. (1994). The fifth discipline: the art and practice of the learning organization. New York: Doubleday.
- Schöckle (1994). An object-oriented environment for modeling and simulation of large continuous systems. (Available at <http://www.nmr.emblheidelberg.de/eduStep/...erences/OOCNS94/Proceedings/Schoeckle.html>).
- Taylor, D. (1990). Object-oriented technology: a manager's guide. Reading: Addison-Wesley.

Vlissides, J., Helm, R., Johnson, R., & Gamma, E. (1995). Design patterns: Elements of reusable object-oriented software. Reading: Addison-Wesley.

Wolstenholme, E. & Corbin, D. (1993). Toward a core set of archetypal structures. Proceedings of the International Systems Dynamics Society. Cancun: System Dynamics Society.